Firedrake

Imperial College London

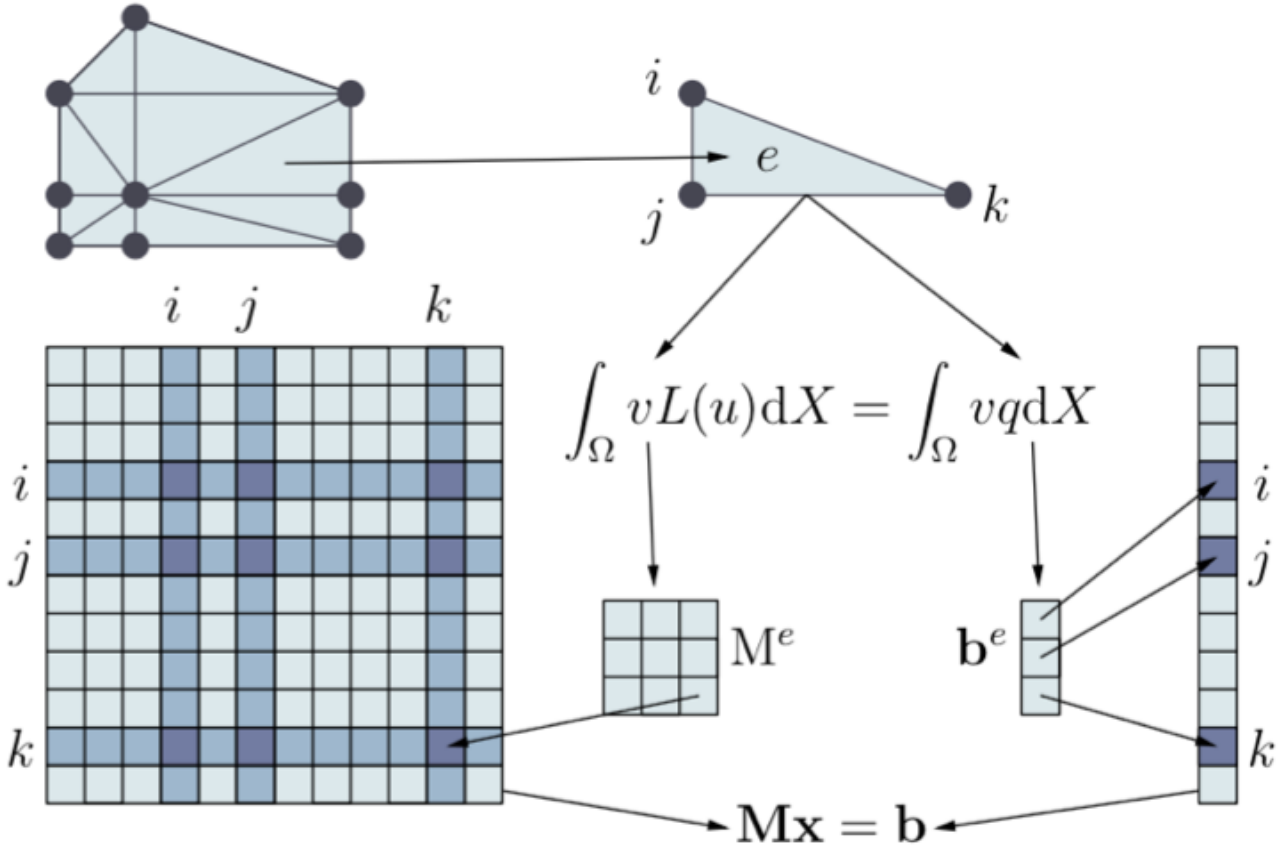
Cross-Element Vectorisation in Firedrake

TJ Sun (ts2914@ic.ac.uk), Lawrence Mitchell, Kaushik Kulkarni, Andreas Klöckner
David A Ham, Paul H J Kelly

April 2019

Finite element method

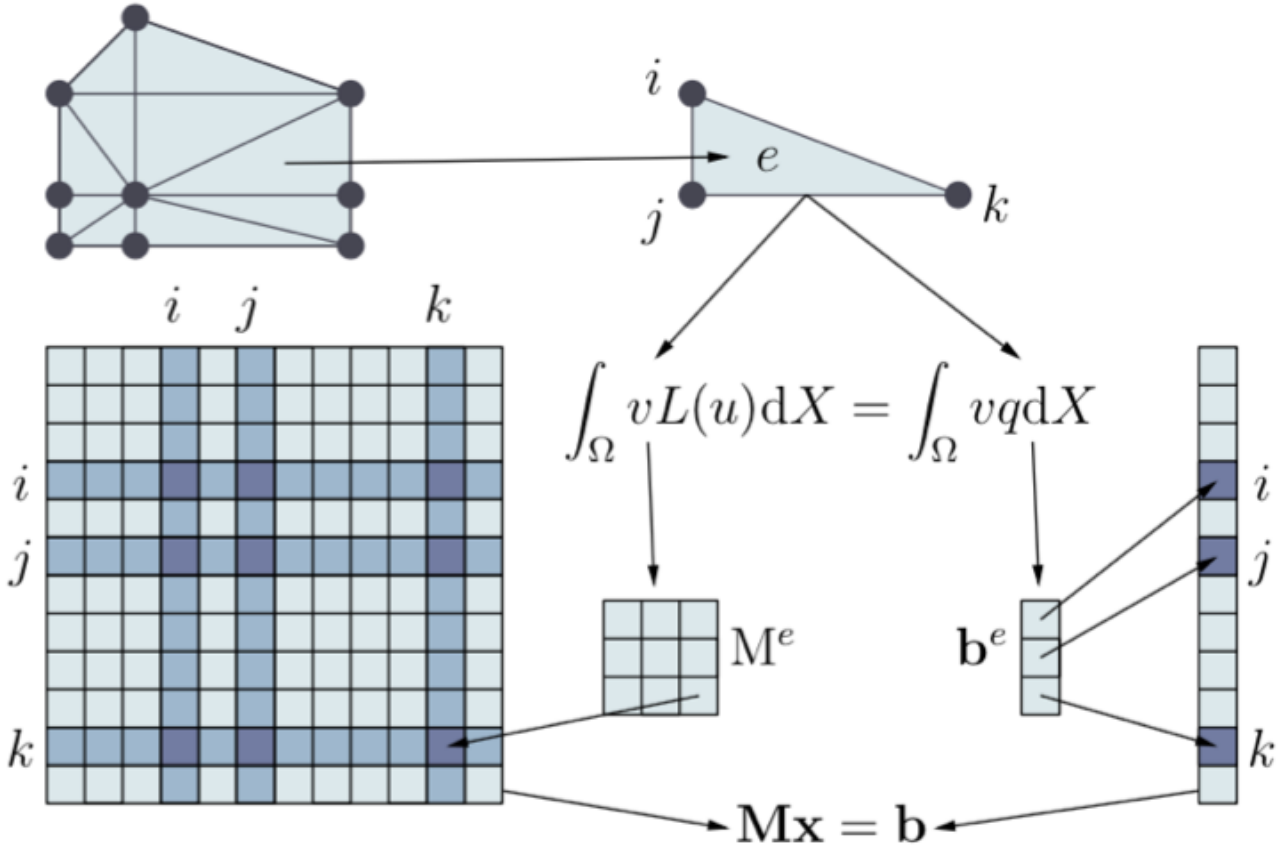
(Computationally) finite element assembly ≈ numerical integration



Rathgeber et al, 2012

Finite element method

(Computationally) finite element assembly ≈ numerical integration



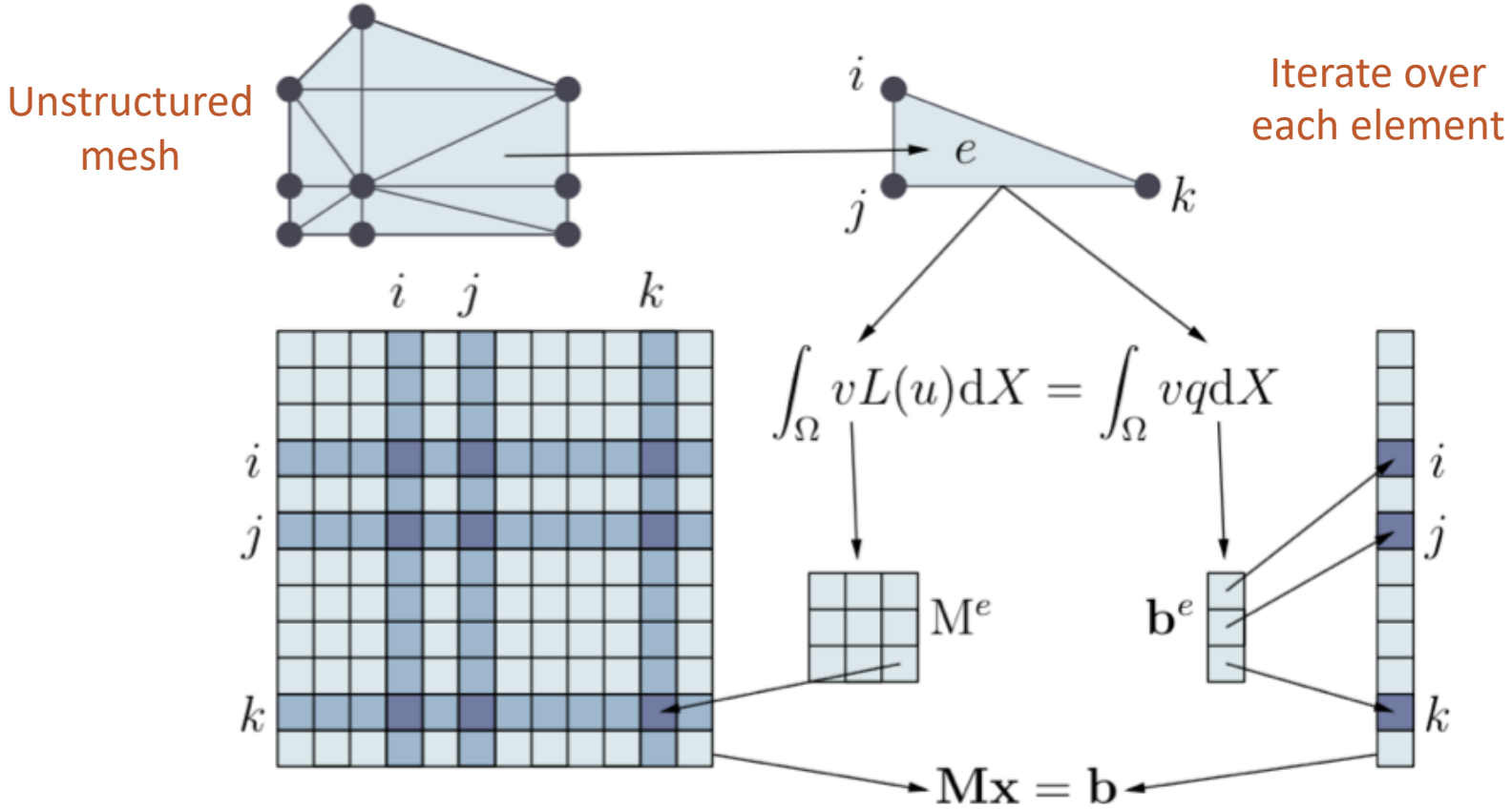
2-form assembly

1-form assembly

Rathgeber et al, 2012

Finite element method

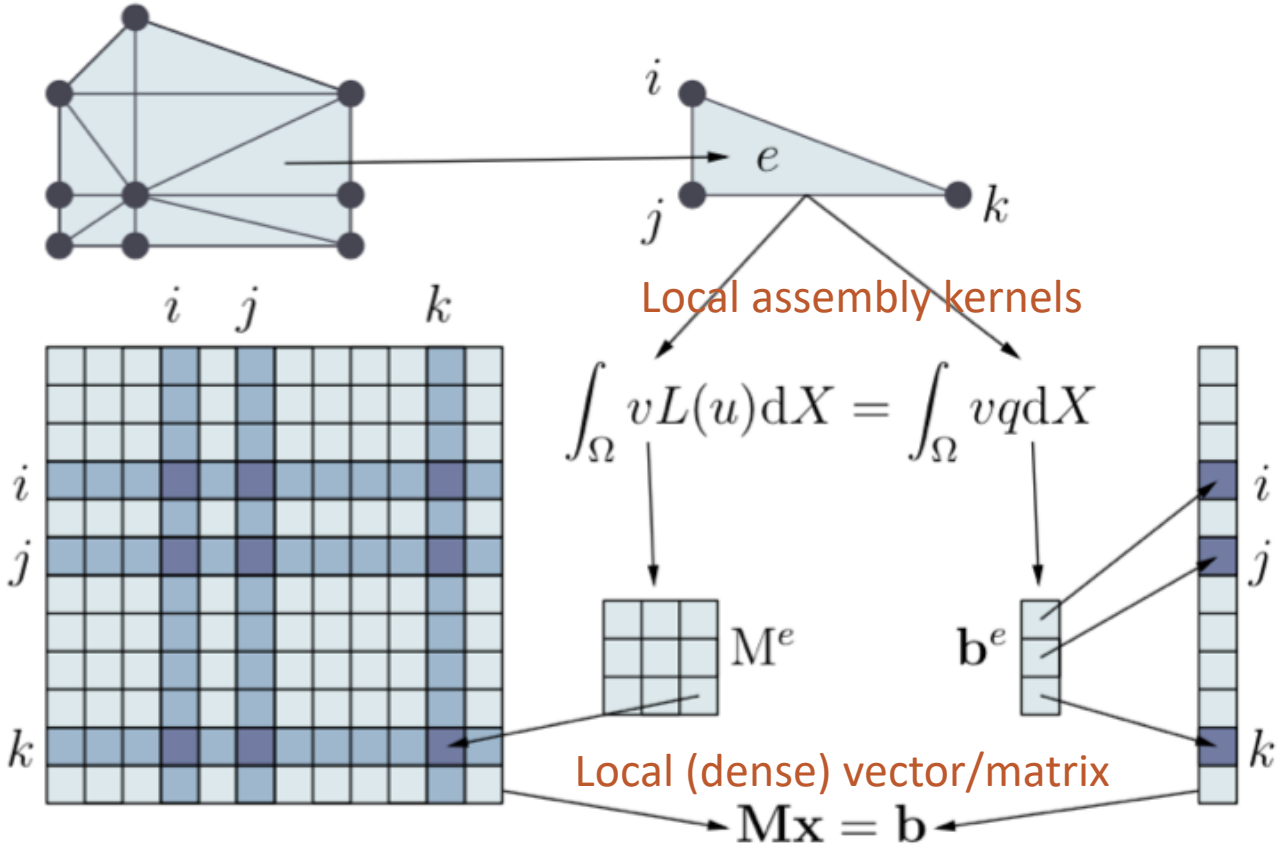
(Computationally) finite element assembly ≈ numerical integration



Rathgeber et al, 2012

Finite element method

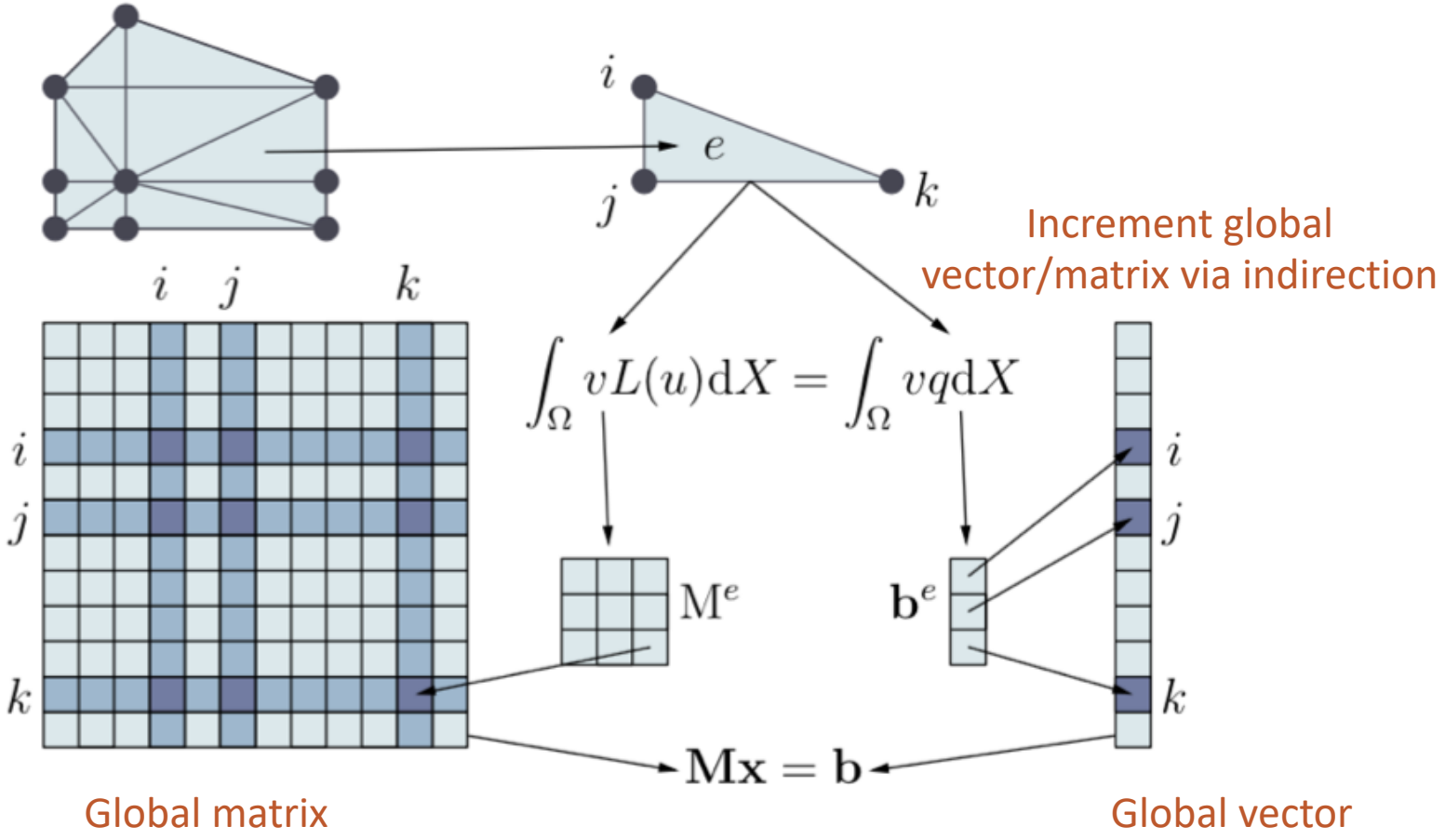
(Computationally) finite element assembly ≈ numerical integration



Rathgeber et al, 2012

Finite element method

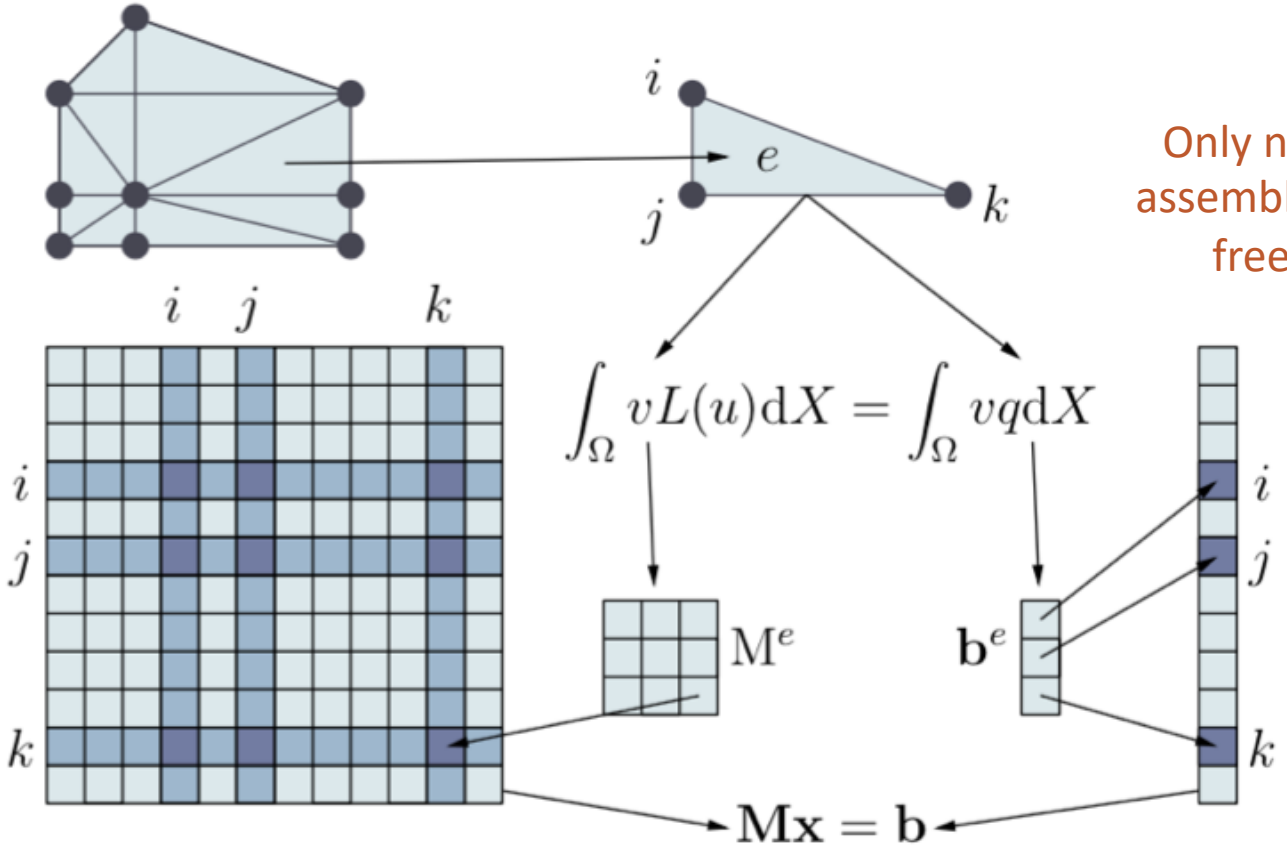
(Computationally) finite element assembly ≈ numerical integration



Rathgeber et al, 2012

Finite element method

(Computationally) finite element assembly ≈ numerical integration



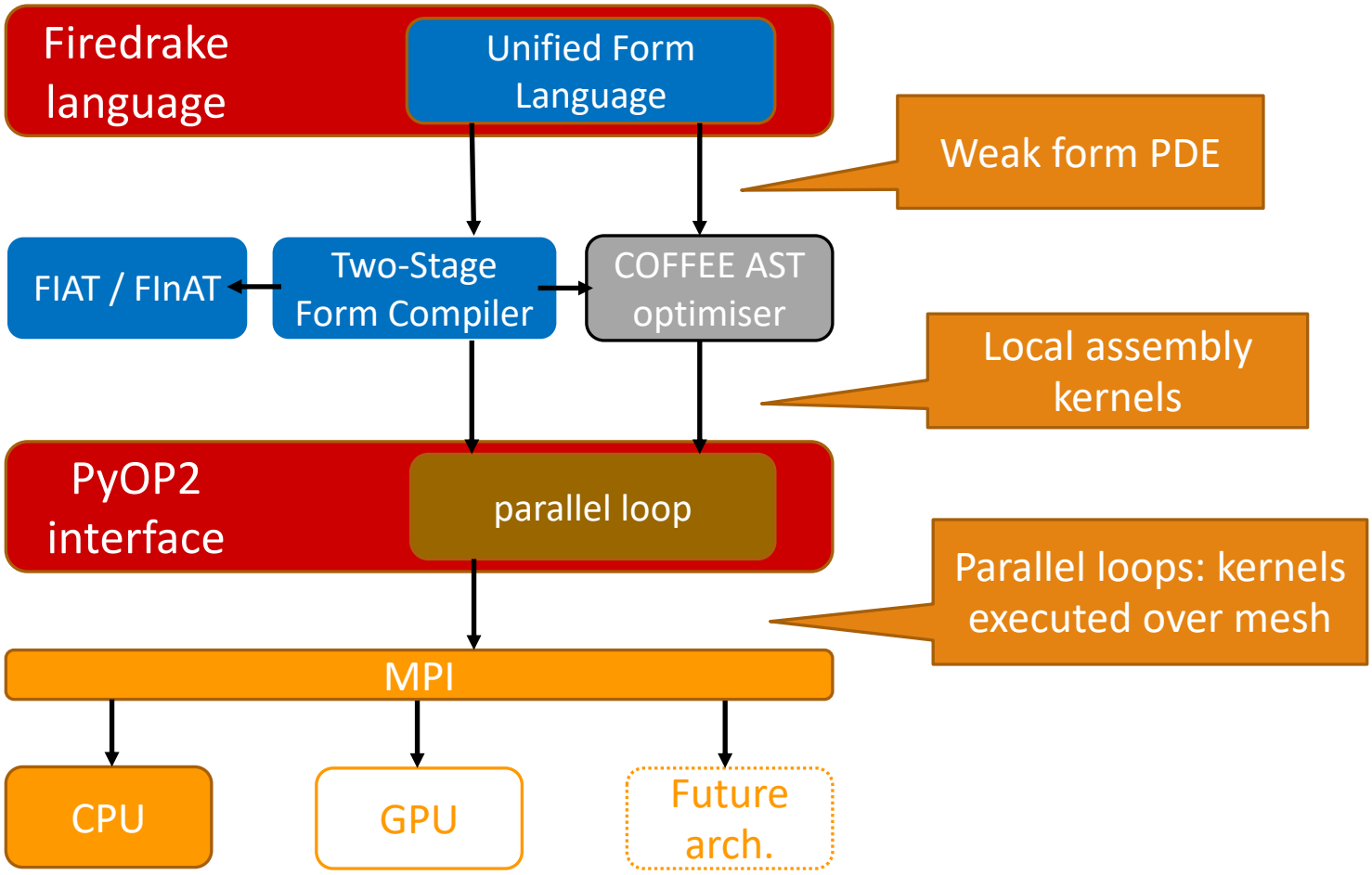
Only need 1-form assembly for matrix-free method

Rathgeber et al, 2012

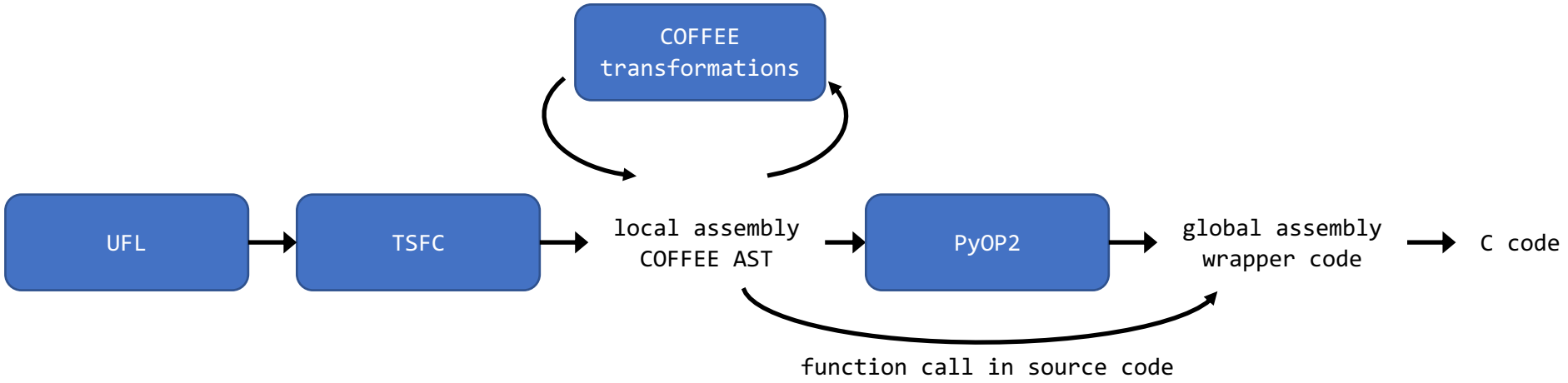
Firedrake example

```
from firedrake import *

mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)


# define the RHS
f = Function(V)
x, y = SpatialCoordinate(mesh)
f.interpolate((1+8*pi*pi)*cos(x*pi*2)*cos(y*pi*2))


# define the PDE
a = (dot(grad(v), grad(u)) + u*v) * dx
L = f * v * dx

u = Function(V)
solve(a == L, u, solver_parameters={'ksp_type': 'cg'})

plot(u)
```

Firedrake overview (simplified)



“Intra-kernel” vectorisation of FE assembly



- Current default in Firedrake
- Vectorisation of local assembly kernel (AST) by COFFEE
- Challenges:
  - Loop trip counts can be small and/or not multiple of SIMD width
  - Usually requires alignment to cache boundary and stride 1 access
  - Operations outside of innermost loop not vectorized
  - Hard to automate for different PDEs, discretisations, meshes

Luporini et al, 2015

Cross-element vectorisation

- Vectorisation of global assembly
- Vector-expand the local assembly kernel to act on N elements together, with N=SIMD width
- All operations can be vectorised
- We can always do this systematically
- But this requires:
  - Intermediate Representation of the whole global assembly
  - Robust loop transformations

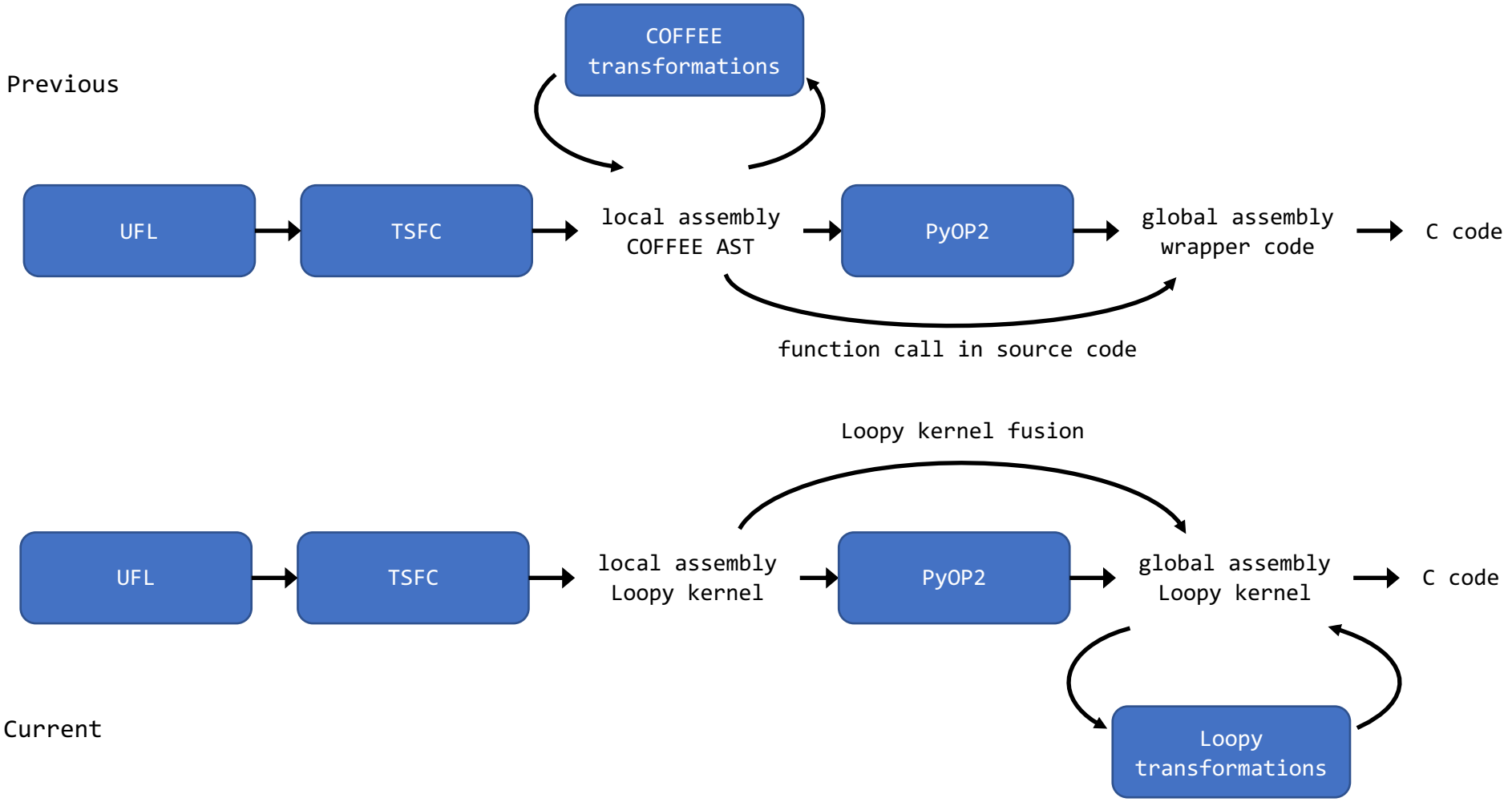
Cross-element vectorisation

- Vectorisation of global assembly
- Vector-expand the local assembly kernel to act on N elements together, with N=SIMD width
- All operations can be vectorised
- We can always do this systematically
- But this requires:
  - Intermediate Representation of the whole global assembly
  - Robust loop transformations

Introducing Loopy as our new code generation backend

- Python package by Andreas Klöckner et al. (UIUC)
- Based on polyhedral model of loops
- ≈ model of loops + transformations
- "Do what I say", not a black-black loop optimiser
- Support multiple backends from same IR: CPU, ISPC, OpenCL, CUDA

Code generation in Firedrake



```
1  // ... Constant array declarations ... //
2
3  void wrap_helmholtz(int const start, int const end, double *__restrict__ dat0, double const
     *__restrict__ dat1, double const *__restrict__ dat2, int const *__restrict__ map0)
4  {
5    double t2[3];
6    double t3[3 * 2];
7    double t4[3];
8
9    for (int n = start; n <= -1 + end; ++n)
10   {
11     for (int i5 = 0; i5 <= 2; ++i5)
12     {
13       t4[i5] = dat2[map0[3 * n + i5]];
14       for (int i6 = 0; i6 <= 1; ++i6)
15         t3[2 * i5 + i6] = dat1[2 * map0[3 * n + i5] + i6];
16     }
17     for (int i1 = 0; i1 <= 2; ++i1)
18       t2[i1] = 0.0;
19
20
21     double form_t11 = 0.0;
22     double form_t1 = -1.0 * t3[0];
23     // ... More similar instructions ... //
24
25     for (int i15 = 0; i15 <= 2; ++i15)
26       dat0[map0[3 * n + i15]] += t2[i15];
27   }
28 }
```

Residual assembly of Helmholtz operator on triangle mesh, CG1 space

```
1  // ... Constant array declarations ... //
2
3  void wrap_helmholtz(int const start, int const end, double *__restrict__ dat0, double const
     *__restrict__ dat1, double const *__restrict__ dat2, int const *__restrict__ map0)
4  {
5    d
6    d
7    d
8
9    for (int n = start; n <= -1 + end; ++n)
10   {
11     for (int i5 = 0; i5 <= 2; ++i5)
12     {
13       t4[i5] = dat2[map0[3 * n + i5]];
14       for (int i6 = 0; i6 <= 1; ++i6)
15         t3[2 * i5 + i6] = dat1[2 * map0[3 * n + i5] + i6];
16     }
17     for (int i1 = 0; i1 <= 2; ++i1)
18       t2[i1] = 0.0;
19
20
21     double form_t11 = 0.0;
22     double form_t1 = -1.0 * t3[0];
23     // ... More similar instructions ... //
24
25     for (int i15 = 0; i15 <= 2; ++i15)
26       dat0[map0[3 * n + i15]] += t2[i15];
27   }
28 }
```

Outer loop over all elements in the mesh

Data gathering

Inlined local kernel

Data scattering

Residual assembly of Helmholtz operator on triangle mesh, CG1 space

The Loopy representation

```
KERNEL:
  helmholtz
ARGUMENTS:
  start: type: int32
  end: type: int32
  dat0: type: float64, shape: (None)
  dat1: type: float64, shape: (None, 2)
  dat2: type: float64, shape: (None)
  map0: type: int32, shape: (None, 3)
DOMAINS:
  [end, start] -> { [n] : start <= n < end }
  { [i2] : 0 <= i2 <= 2 }
  // ... More domains ... //
INAME_IMPLEMENTATION_TAGS:
  None
TEMPORARIES:
  t4: type: float64, shape: (3), dim_tags: (stride:1)
  // ... More temporaries ... //
```

```
INSTRUCTIONS:
 for n, i2
     t4[i2] = dat2[map0[n, i2]]
     for i3
       t1[i2, i3] = dat1[map0[n, i2], i3]
   end i2, i3
   for i0
     t0[i0] = 0
   end i0
   // ... Inlined local assembly kernel ... //
   form_t11 = 0.0
   // ... More instructions ... //
   for i15
     dat0[map0[n, i15]] += t0[0, i15]
 end n, i15
```

The Loopy representation

```
KERNEL:
  helmholtz
ARGUMENTS:
  start: type: int32
  end: type: int32
  ape: (None)
  ape: (None, 2)
  ape: (None)
  map0: type: i  shape: (None, 3)
DOMAINS:
  [end, start] -> { [n] : start <= n < end }
  { [i2] : 0 <= i2 <= 2 }
  // ... More domains ... //
INAME_IMPLEMENTATION_TAGS:
  None
TEMPORARIES:
  t4: type: float64, shape: (
  // ... More temporaries ... //
```

Loop indices are called “inames”

Affine inequalities describe the loop bounds

Tags governs code generation behaviour

```
INSTRUCTIONS:
 for n, i2
     t4[i2] = dat2[map0[n, i2]]
     for i3
       t1[i2, i3] = dat1[map0[n, i2], i3]
   end i2, i3
   for i0
     t0[i0] = 0
                                  assembly kernel ... //
   // ... More instructions ... //
   for i15
     dat0[map0[n, i15]] += t0[0, i15]
 end n, i15
```

Cross-element vectorisation by applying Loopy transformations

1. Split the outer loop n into n_outer and n_simd
   - With n_simd extent = SIMD width
2. Tag n_simd with implementation tag “SIMD”

```
// ... Constant array declarations ... //

void wrap_helmholtz(int const start, int const end, double *__restrict__ dat0, double const *
  __restrict__ dat1, double const *__restrict__ dat2, int const *__restrict__ map0)
{
  double t2[3 * 4] __attribute__ ((aligned (64)));
  // ... More temporary array declarations ... //
  for (int n_out = (start / 4); n_outer <= ((end-4) / 4); ++n_out) {
    for (int i5 = 0; i5 <= 2; ++i5) {
      for (int i6 = 0; i6 <= 1; ++i6) {
        #pragma omp simd
        for (int n_simd = 0; n_simd <= 3; ++n_simd)
          t3[n_simd+8*i5+4*i6]=dat1[2*map0[12*n_out+3*n_simd+i5]+i6];
      }
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t4[n_simd+4*i5] = dat2[map0[12*n_out+3*n_simd+i5]];
    }
    for (int i1 = 0; i1 <= 2; ++i1) {
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t2[n_simd + 4 * i1] = 0.0;
    }
    #pragma omp simd
    for (int n_simd = 0; n_simd <= 3; ++n_simd) {
      form_t11[n_simd] = 0.0;
      form_t1[n_simd] = -1.0 * t3[n_simd];
      // ... More similar instructions ... //
    }
    for (int i15 = 0; i15 <= 2; ++i15)
      for (int n_batch = 0; n_batch <= 3; ++n_batch)
        dat0[map0[12*n_out+3*n_batch+i15]] += t2[n_batch+4*i15];
  }
}
```

Residual assembly of Helmholtz operator on triangle mesh, CG1 space, batched by 4

Temporaries vector-expanded, with n_simd dimension fastest moving

n_simd loops pushed to innermost, decorated by pragma

Write-back is serialised, due to potential race condition

```
// ... Constant array declarations ... //
__restrict__ dat1, double const *__restrict__ dat2, int const *__restrict__ map0)
{
  double t2[3 * 4] __attribute__ ((aligned (64)));
  // ... More temporary array declarations ... //
  for (int n_out = (start / 4); n_outer <= ((end-4) / 4); ++n_out) {
    for (int
      for (in
        #pragma omp simd
        for (int n_simd = 0; n_simd <= 3; ++n_simd)
          t3[n_simd+8*i5+4*i6]=dat1[2*map0[12*n_out+3*n_simd+i5]+i6];
      }
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t4[n_simd+4*i5] = dat2[map0[12*n_out+3*n_simd+i5]];
    }
    for (int i1 = 0; i1 <= 2; ++i1) {
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t2[n_simd + 4 * i1] = 0.0;
    }
    #pragma omp simd
    for (int n_simd = 0; n_simd <= 3; ++n_simd) {
      form_t11[n_simd] = 0.0;
      form_t1[n_simd] = -1.0 * t3[n_simd];
      // ... Mo
    }
    for (int i1
      for (int n_batch = 0; n_batch <= 3; ++n_batch)
        dat0[map0[12*n_out+3*n_batch+i15]] += t2[n_batch+4*i15];
  }
}
```

Residual assembly of Helmholtz operator on triangle mesh, CG1 space, batched by 4

Compiler vector extensions

- A more direct way to inform compiler to generate SIMD instructions than using OpenMP pragma
- For example
  - typedef double double4 __attribute__ ((vector_size (32)));
- We need to add new features to Loopy
  - New code generation target with vector data types
  - New iname tag

```
typedef double double4 __attribute__ ((vector_size (32)));
typedef int int4 __attribute__ ((vector_size (16)));

// ... Constant array declarations ... //

void wrap_form0_cell_integral_otherwise(int const start, int const end, double *__restrict__ dat0,
  double const *__restrict__ dat1, double const *__restrict__ dat2, int const *__restrict__ map0)
{
  double4 form_t1 __attribute__ ((aligned (64)));
  // ... Temporary array declarations ... //

  for (int n_outer = (start / 4); n_outer <= ((-4 + end) / 4); ++n_outer) {
    for (int i5 = 0; i5 <= 2; ++i5) {
      for (int i6 = 0; i6 <= 1; ++i6) {
        #pragma omp simd
        for (int n_simd = 0; n_simd <= 3; ++n_simd)
          t3[(2 * i5 + i6)][n_simd] = dat1[2 * map0[12 * n_outer + 3 * n_simd + i5] + i6];
      }
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t4[i5][n_simd] = dat2[map0[12 * n_outer + 3 * n_simd + i5]];
    }
    for (int i1 = 0; i1 <= 2; ++i1)
      t2[i1] = _zeros_double4;

    form_t11 = _zeros_double4;
    form_t1 = -1.0 * t3[0];
    // ... More similar instructions ... //

    for (int i15 = 0; i15 <= 2; ++i15)
      for (int n_batch = 0; n_batch <= 3; ++n_batch)
        dat0[map0[12 * n_outer + 3 * n_batch + i15]] += t2[i15][n_batch];
  }
}
```

Residual assembly of Helmholtz operator on triangle mesh, CG1 space, vector extensions

```
1  typedef double double4 __attribute__ ((vector_size (32)));
2  typedef int int4 __attribute__ ((vector_size (16)));
3
4  // ... Constant array declarat
5
6  void wrap_form0_cell_integral_otherwise(int const start, int const end, double *__restrict__ dat0,
     double const *__restrict__ dat1, double const *__restrict__ dat2, int const *__restrict__ map0)
7  {
8    double4 form_t1 __attribute__ ((aligned (64)));
9    // ... Temporary array declarations ... //
10
11   for (int n_outer = (start / 4); n_outer <= ((-4 + end) / 4); ++n_outer) {
12     for (int i5 = 0; i5 <= 2; ++i5) {
13       for (int i6 = 0; i6 <= 1; ++i6) {
14         #pragma omp simd
15         for (int n_simd = 0; n_simd <= 3; ++n_simd)
16           t3[(2 * i5 + i6)][n_simd] = dat1[2 * map0[12 * n_outer + 3 * n_simd + i5] + i6];
17       }
18       #pragma omp simd
19       for (int n_simd = 0; n_simd <= 3; ++n_simd)
20         t4[i5][n_simd] = dat2[map0[12 * n_outer + 3 * n_simd + i5]];
21     }
22     for (int i1 = 0; i1 <= 2; ++i1)
23       t2[i1] = _zeros_double4;
24
25     form_t11 = _zeros_double4;
26     form_t1 = -1.0 * t3[0];
27     // ... More similar instructions ... //
28
29     for (int i15 = 0; i15 <= 2; ++i15)
30       for (int n_batch = 0; n_batch <= 3; ++n_batch)
31         dat0[map0[12 * n_outer + 3 * n_batch + i15]] += t2[i15][n_batch];
32   }
33 }
```

Vector data types

Vector instructions

Residual assembly of Helmholtz operator on triangle mesh, CG1 space, vector extensions

Experimental setup

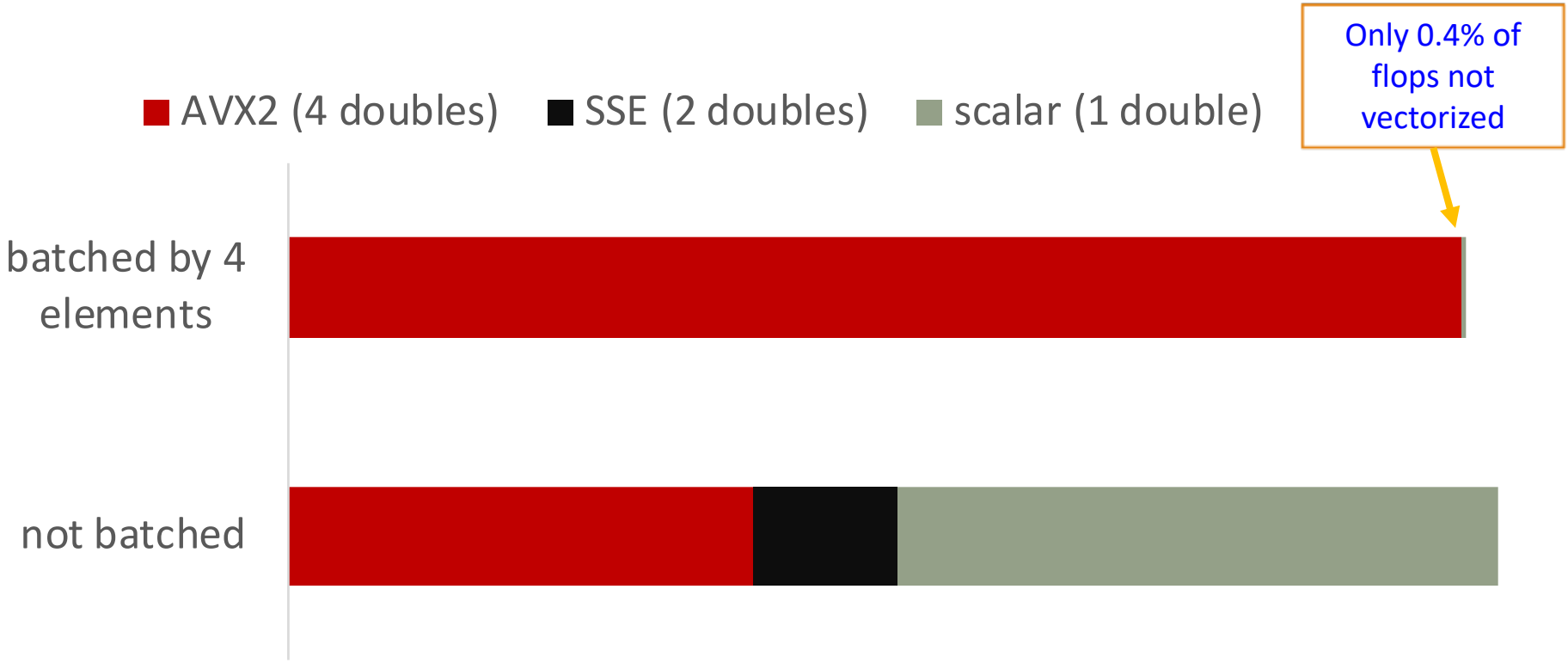
- Hardware

	Haswell Xeon E5-2640 v3	Skylake Xeon Gold 6148
Base frequency	2.6 GHz	2.4 GHz
Physical cores	8	20
SIMD instruction set	AVX2	AVX512
`doubles` per SIMD vector	4	8
Cross-element vectorization batch size	4	8
FMA[4] units per core	2	2
FMA instruction issue per cycle	2	2
Peak performance (double-precision)[5]	332.8 GFLOP/s	1536.0 GFLOP/s
LINPACK performance (double-precision)[6]	262.5 GFLOP/s	976.7 GFLOP/s
Memory bandwidth[7]	38.5 GB/s	81.0 GB/s

- We measure all cores in a single node and report % of peak performance
- 4 meshes: triangles, quadrilaterals, tetrahedra, hexahedra
- 5 operators
- 3 compilers: GCC, ICC, Clang

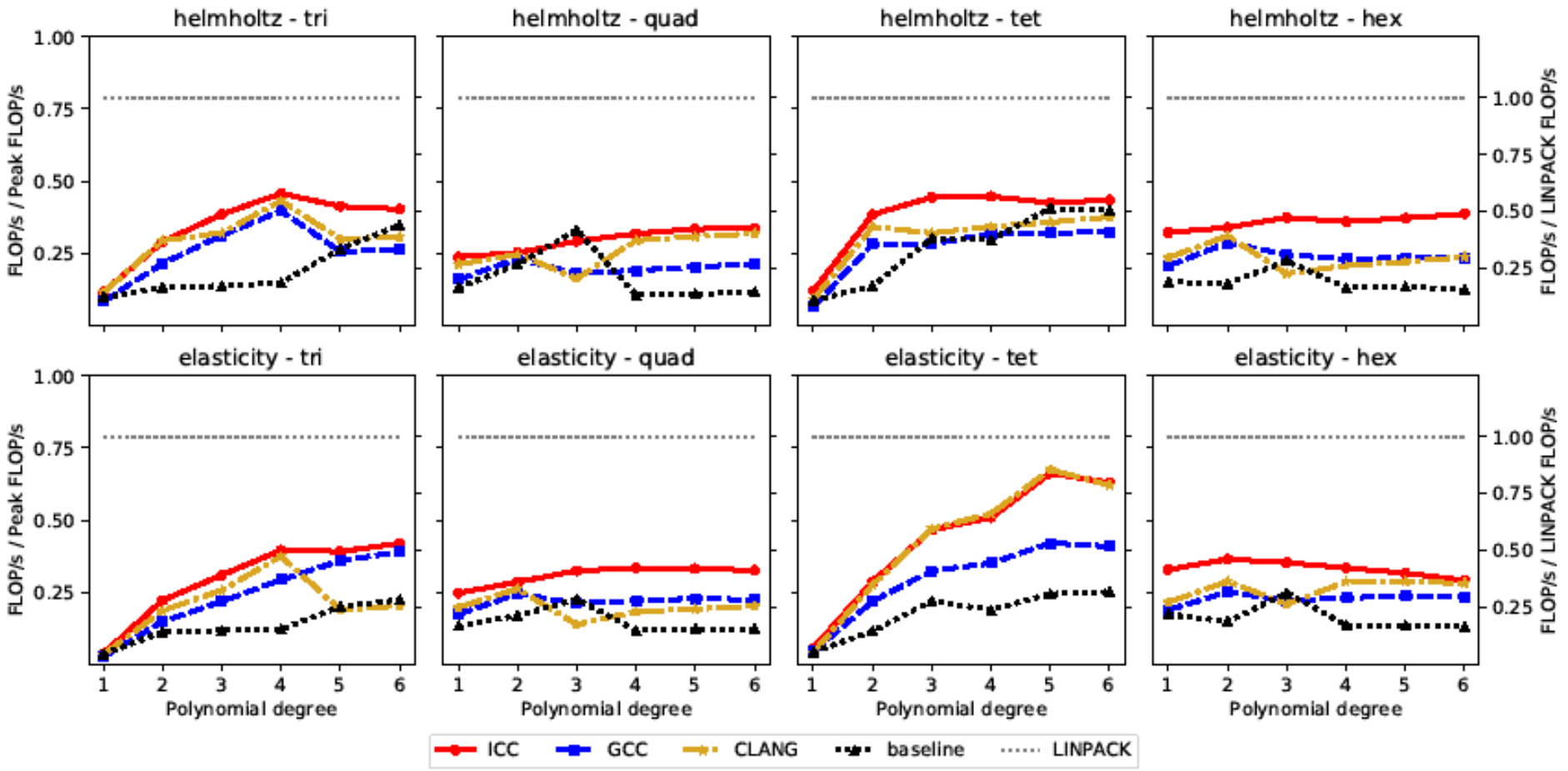
Check that SIMD instructions are generated

- Flop contributions by instruction types (Helmholtz on hexahedra, degree 4, ICC)



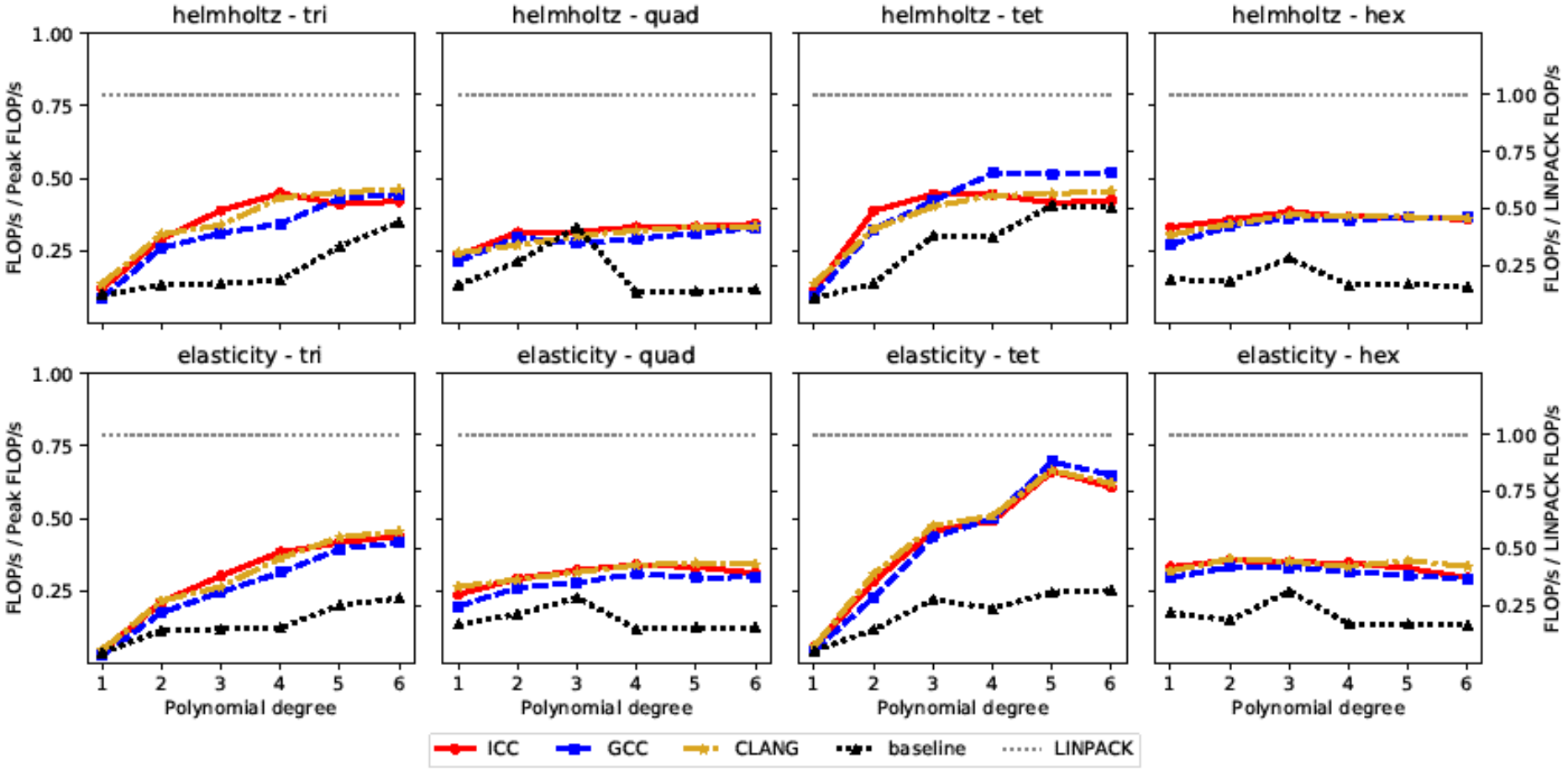
Haswell, OpenMP pragma

- ICC is consistent across different polynomial degrees



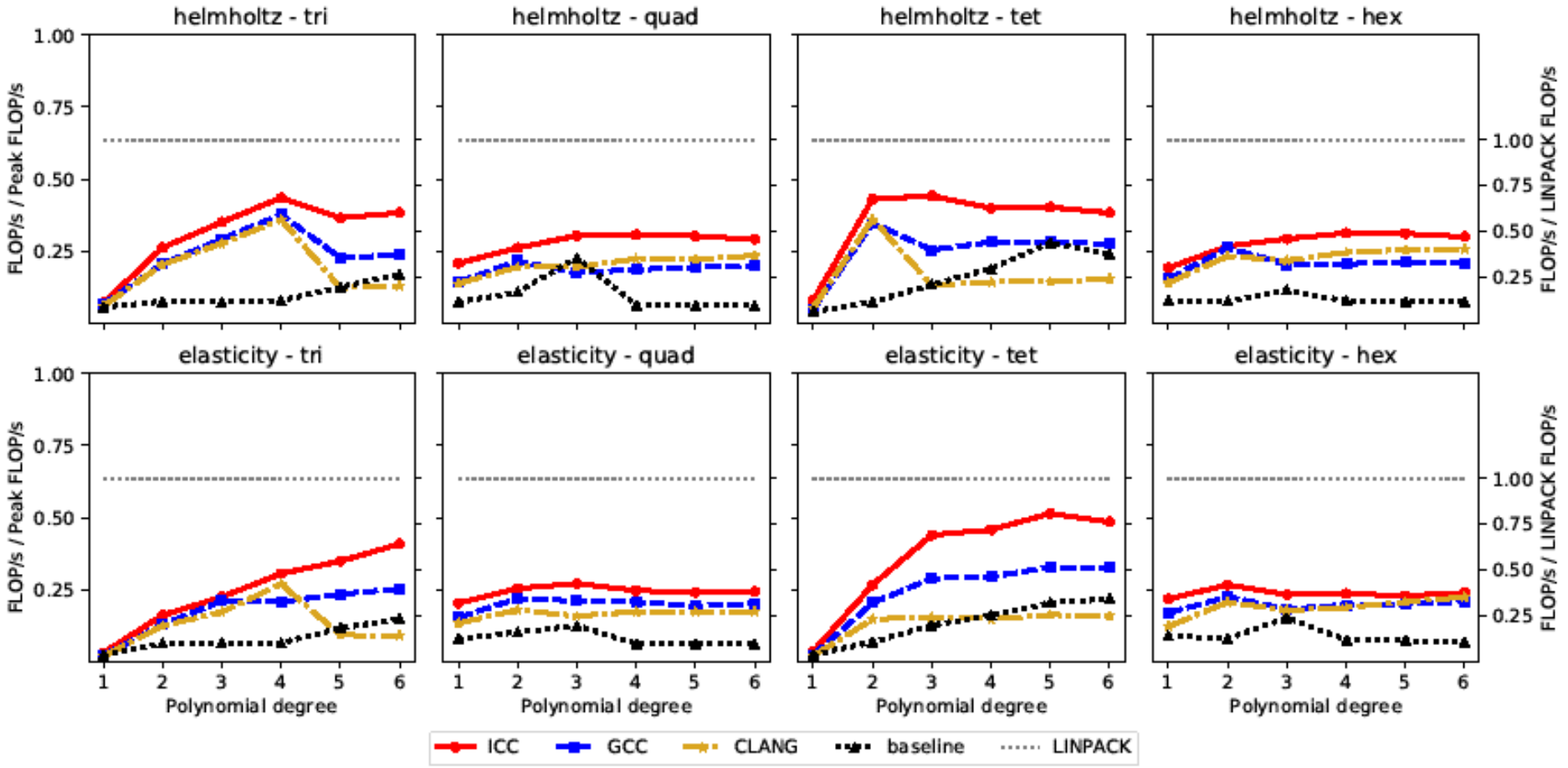
Haswell, vector extensions

- Vector extensions give portable performance



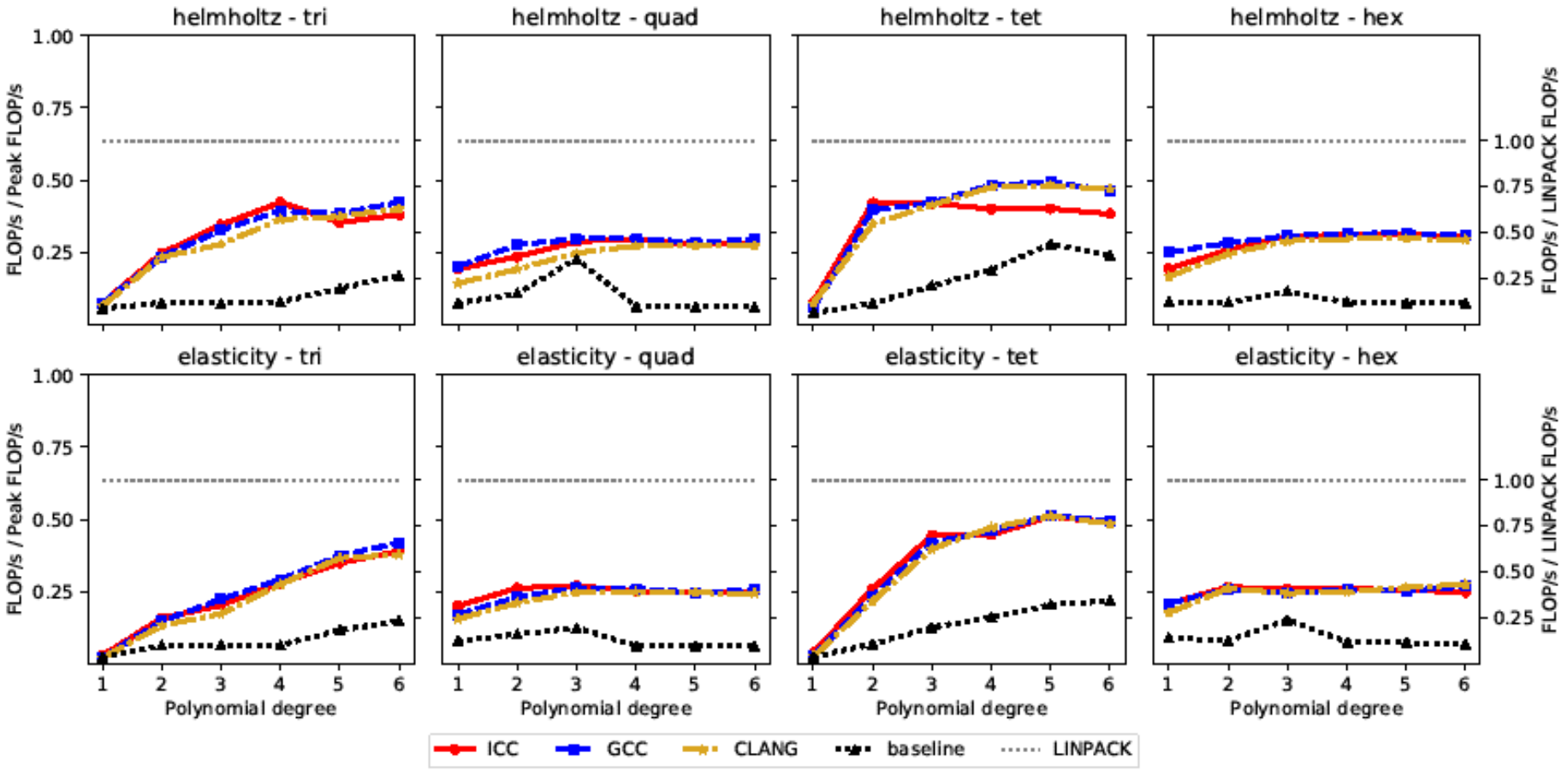
Skylake, OpenMP pragma

- Similar observations as Haswell

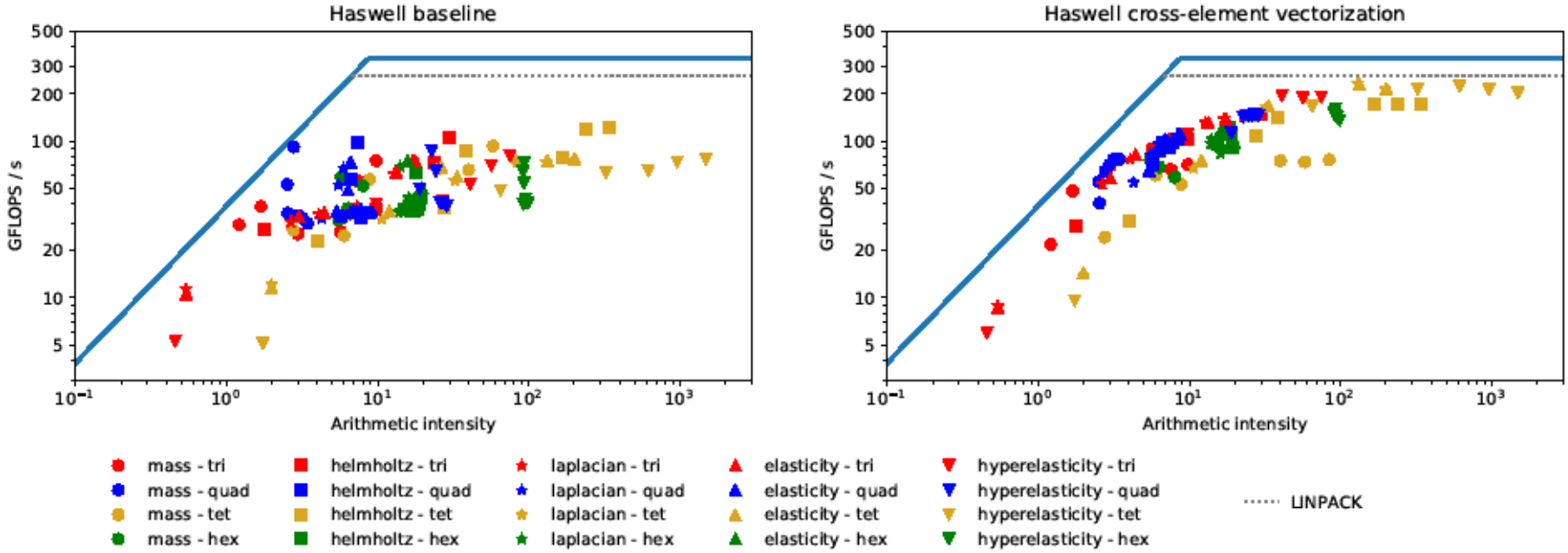


Skylake, vector extensions

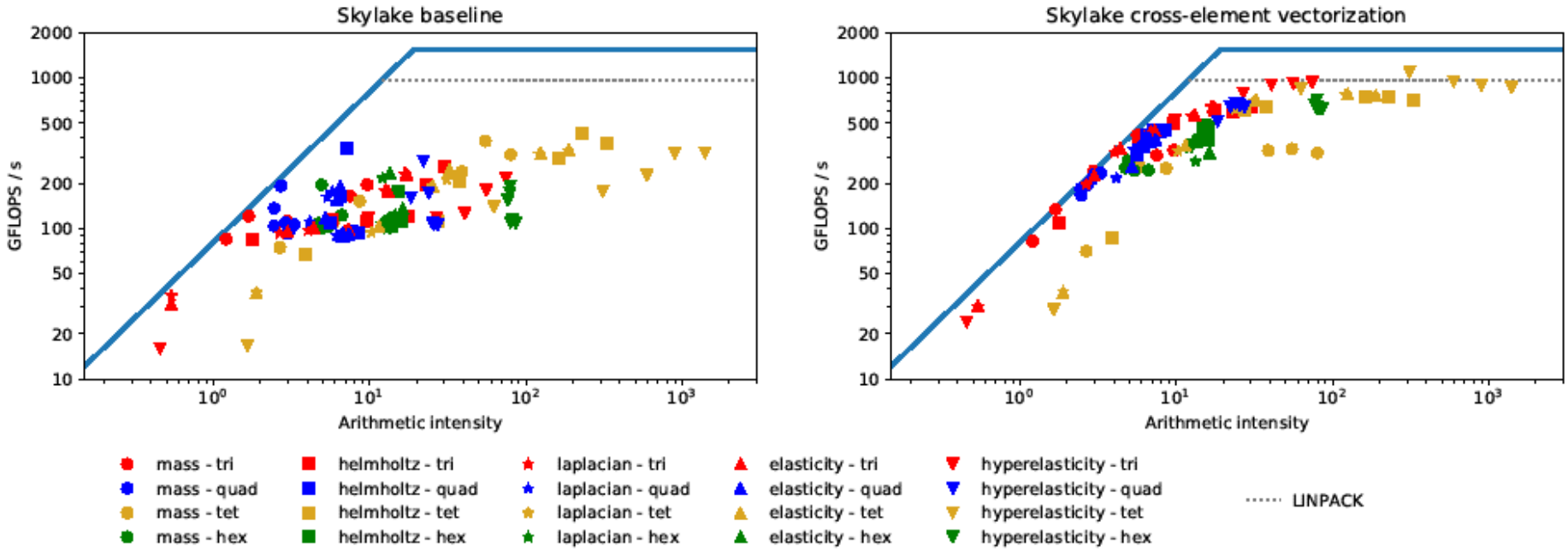
- Similar observations as Haswell



Haswell roofline



Skylake roofline



Conclusion

- Cross-element vectorisation consistently achieves >30% peak performance for residual assembly
- New abstraction for global assembly enables automated and robust transformation and code generation, powered by Loopy
- More to come…
  - Better support for bilinear form assembly (currently need to pack into individual element matrices and call PETSc function MatSetValutes)
  - Pathway to GPUs (explore more vectorisation strategies other than “1 element per thread”)
- Preprint on arXiv: A study of vectorization for matrix-free finite element methods
- www.firedrakeproject.org for docs, demos, notebooks

Firedrake